Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Tutorial Two

Creating the Structure

Contents

Introduction 3
Tutorial Requirements 3
Part One: Defining the data structure 4
Part Two: Defining the data file structure 8
Part Three: Using the Data File 10

Introduction

In tutorial one, we learned how to create a new C# project, how to create a windows menu and attach code to the menu items to get them to do specific things when clicked, and how to run and test our program.

We also decided on a basic structure for the content we wish to hold, and during this tutorial, we will define the data structures that our program (UCL) will need in order to use this content, along with a file format required for storing them.

Tutorial Requirements

These tutorials are sequential, so if you haven't already been through the previous tutorial, then please download and work through it before continuing with this one.

Tutorial one also outlines the software you will need for creating UCL and where you can download it for free.

Part One: Defining the data structure

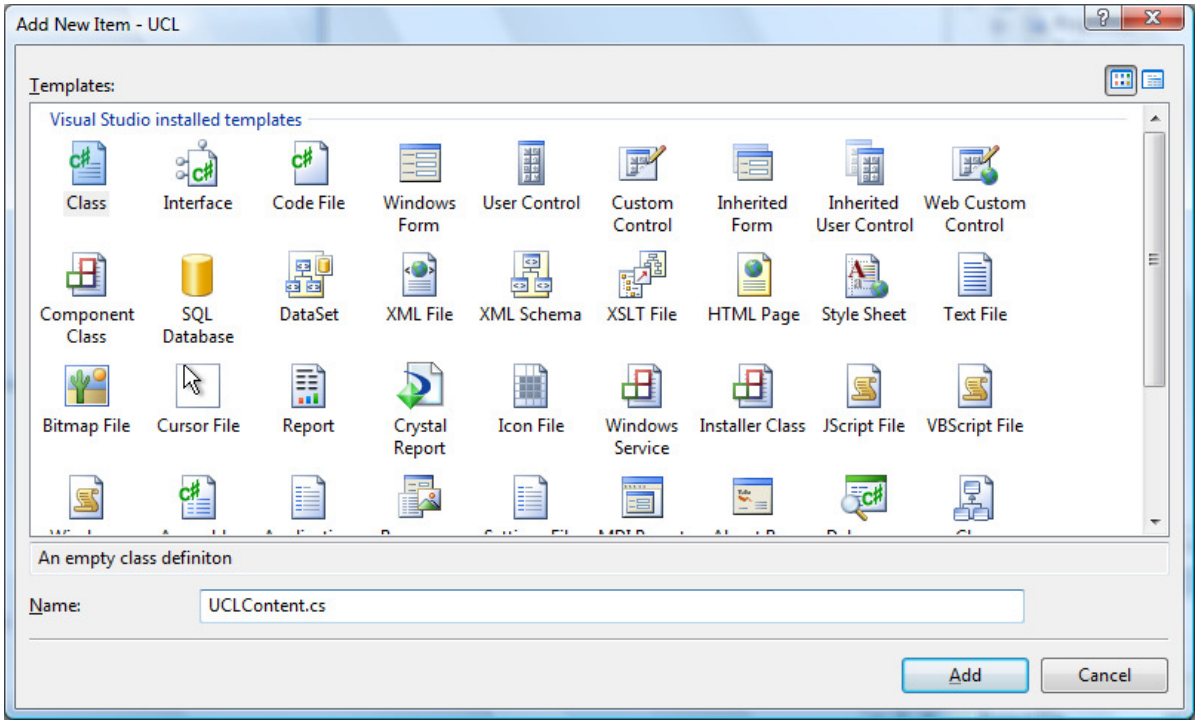
Cast your mind back to tutorial one where we first defined the data structure we want to use for holding content information in UCL. Here is what we said we would start with:

- Type
- Name
- Icon
- ExeName
- ExePath
- FileName
- FilePath
- Arguments

To create this data structure in UCL we are going to define a new class. We will use a class, as opposed to a Struct, as we will also need to eventually define methods for our content, such as a Launch() method to actually get our content to run from within UCL.

To create a new class, right click on your project name (UCL) in the solution explorer, then select Add> and Class.

You will then see the template screen as shown below:



The Class option in the Template window is already pre-selected, so just give the new class a name – UCLContent.cs and click on the Add button. You should now see your new UCLContent class code in the main window, note that UCLContent.cs is now also listed in the solution explorer along with your other project files.

The UCLContent.cs class already has some code created for us that looks like this:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace UCL
{
    class UCLContent
    {
    }
}
```

No surprises here, as we already covered the meaning of all that code in the first tutorial, so let's start defining our class by typing in between the braces for the `class UCLContent` section. If you recall, we said that most of the data we need to hold will be of a type called `string`, so let's enter them now. To do this, we simply need to type "string", followed by a space and the name we want to give to our data. Don't forget the ";" at the end of each line:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace UCL
{
    class UCLContent
    {

        string Name;
        string Icon;
        string ExeName;
        string ExePath;
        string FileName;
        string FilePath;
        string Arguments;

    }
}
```

So, why have we not put in a `string` for the content Type? Well there's a better way of storing this in C# that will be more efficient, and even allow us to easily select all the different choices for the Type from within the IDE's context menus at a later date. This is a type of structure known in C# as an `enum`, and we can create it like so:

```
        enum UCLType
        {

        }
```

An enum is special, as it allows us to create a list of items and assign a value to each of these. So we could have an enum containing Game, Application, Video and Audio and these would be given the values of 0, 1, 2 and 3 respectively. This means that when we create some new content with the type Video, C# will assign the value 2 to the Type for that content, which is far more efficient for C# to use, but it will actually display the name for the Type (Video) in the IDE, which is easier for us to use!

You can override the automatic values that C# gives to an enum item by also including a value for it in the definition. So using the above example, we could enter this as our Type enum:

```
enum UCLType
{
    Game = 0,
    Application = 1,
    Video = 2,
    Audio = 3,
}
```

Now, notice that we don't use semi-colons when defining an enum. This is because a semi-colon tells C# to process the line and move on to the next, and we need the enum to be processed all in one go, so we use a comma to separate the types in our list. We also chose to assign a value for each type in the example above, we could have excluded this, and the end result would have been the same, as enum values will start at zero and increment.

The nice thing about creating an enum is that we don't have to worry about adding additional types at a later date, the enum can be easily modified, simply by extending the list, and any additional values will be instantly available throughout our code, so we can simply leave it at that for now.

Note that we have defined separate values for a Game and Application in our enum. If you think about it, a Game and Application will probably be launched in exactly the same way from UCL by simply running their executable file, but we have defined them as separate types to allow us to filter one or the other at a later date. We have therefore made it possible to add something like a tab control in the future that could easily display Games in one tab and Applications in the other.

So, enter this enum directly above the "`string Name;`" line in our UCLContent class. What we are actually doing here is creating an enum called UCLType that can be the value of Game, Application, Video or Audio, but we haven't yet created anything to hold this type value within our class, the enum statement only defines the structure of the type itself.

If we look at Name, it is of type string, and we can create a Type for our enum in exactly the same way. So below the enum statement, and above the Name declaration, enter the following:

```
UCLType Type;
```

Now we have a `Type` for our UCLContent class that we can set to any value that is defined in the `enum UCLType`. This will become very clear to you when we actually start using the class. At the end of all that, you should end up with a UCLContent class that looks like the following:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace UCL
{
    class UCLContent
    {

        enum UCLType
        {
            Game = 0,
            Application = 1,
            Video = 2,
            Audio = 3,
        }

        UCLType Type;
        string Name;
        string Icon;
        string ExeName;
        string ExePath;
        string FileName;
        string FilePath;
        string Arguments;

    }
}
```

Now we've typed all that in, let's not lose it, so click on the save button (the blue disk image) in the IDE toolbar. Note that you can also click on the "Save All" button (the multiple disks image) to save all files that have changed in your project.

Now, if you know a bit about programming with classes, then you may see a problem with the class we have defined.

It is important though that you understand what we did here, we have written a perfectly valid class, it's just that later in our project; we will come across some issues with using it. You will learn far more by fixing these issues (as and when we find them) than if I simply told you what a working class file for this project should look like, as that takes insight and experience, which you simply don't have when starting out in a new programming language.

For now, it is a good start, and at this point, with our limited knowledge of C#, it seems to be up to the job at hand! Make sure you use the class definition above in your project, even if you think you know how to fix it, because the issues we will face later will raise a number of errors based on this definition.

Part Two: Defining the data file structure

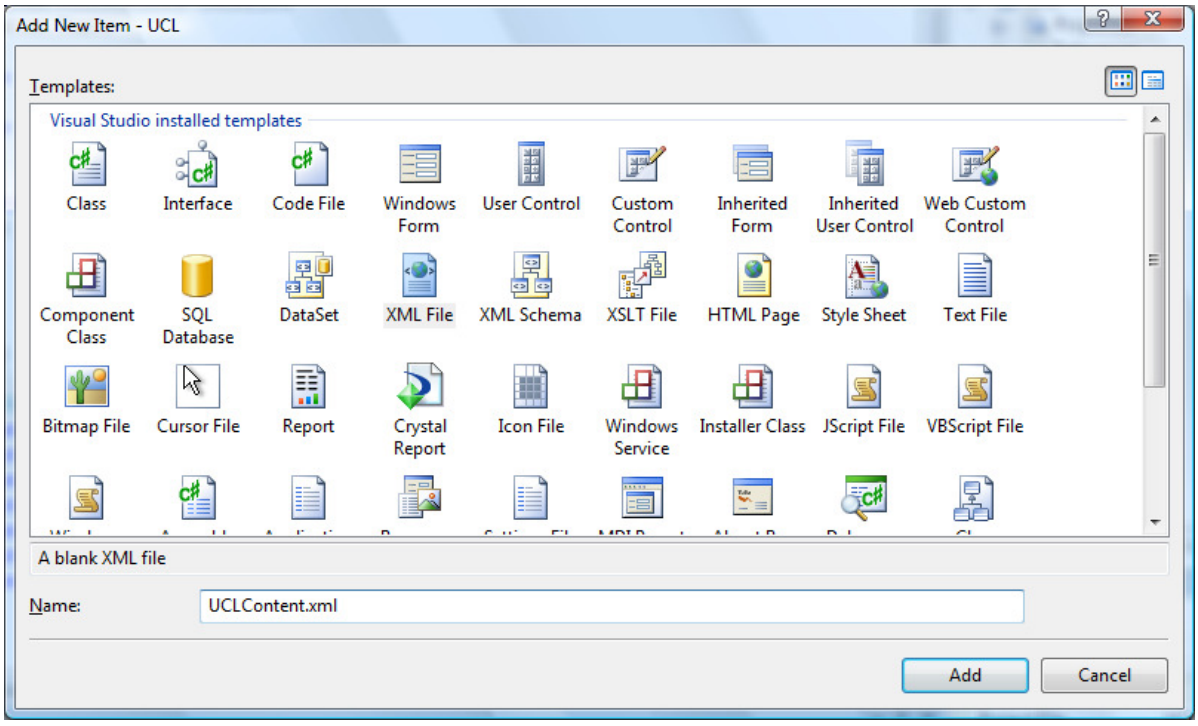
Now we have the data structure sorted, we need to create a file structure to store the data when our program is not running. This will allow us to load in the data file when our program starts, update it whenever there are any changes, and by doing that, our data won't get lost when we exit the program as it will be stored on the computers local disk.

Traditionally this information would be stored in an "ini" file format, which has become a popular choice for windows based applications. However, as UCL is a fresh and modern launcher, we need a fresh and modern data file storage solution, and there is one that fits the requirements perfectly – XML.

Using XML we can do all of the above, but we also get some added bonuses in the way that it is structured, that will allow us to load our data easily when we define our content list in UCL. Now, ideally we would go off now and create some new forms that will allow us to populate such a file with the content we want to launch, however that would be like running before we can walk, so instead we will look at the structure of an XML file, and then manually create a test file for a few content items that we can go on to display in UCL. After this is working, we can then address the issue of how we can modify the content data file from directly within UCL.

So, to start with let's create a blank data file. In the solution explorer, right click on your project (UCL) and select Add> New Item...

You will see the template screen again, like the one we saw when we created a new class file, but this time we want to select an XML file and give it the name UCLContent.xml as shown below:



You will notice that your new XML file is displayed with the following line already present:

```
<?xml version="1.0" encoding="utf-8" ?>
```

We don't need to get hung up about this, as we're going to leave it alone anyway. But needless to say your file needs this to function properly as it defines the version of XML it is compliant with and the encoding method that it uses. Let's just ignore it for now and continue, as for this project we simply do not need to change this.

First we need to create what is called a root element; this will contain all the other elements in our XML file in one big collection, a bit like a treeview! Let's call this root element "content". In XML we need to define a new element by enclosing it between "<" and ">" so type in "<content>" directly below the XML version line shown above, in your UCLContent.xml file, and then press enter.

You will notice that the IDE automatically added </content> after the text that you typed. If you've written HTML before then this will seem familiar, if you haven't then just remember that every element in XML must have a matching pair of tags to work correctly. Luckily our IDE makes them hard to miss by creating the matching pair for us!

It is also worth noting at this point that XML elements, like C# objects, are case sensitive, so you need to be a little bit careful of what you are typing in.

Now what I want you to do is define an entry for an application that is installed on your computer. It will be easier, for the purposes of testing, if this is a small application that loads quickly. Here is an example for my computer whereby I have simply defined Microsoft Notepad as an entry in the XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<Content>
  <Application>
    <Name>Microsoft Notepad</Name>
    <ExeName>notepad.exe</ExeName>
    <ExePath>c:\windows\</ExePath>
  </Application>
</Content>
```

Note that the element names I have defined for <Application> match the names we setup in our UCLContent class in part one. Where I have an appropriate value, then I have entered it in the example above, where I do not know the value, I have excluded it, so for starters my first application only has "Name", "ExeName" and "ExePath" defined. I would like you to do the same for an application that is on your computer with just those three elements included.

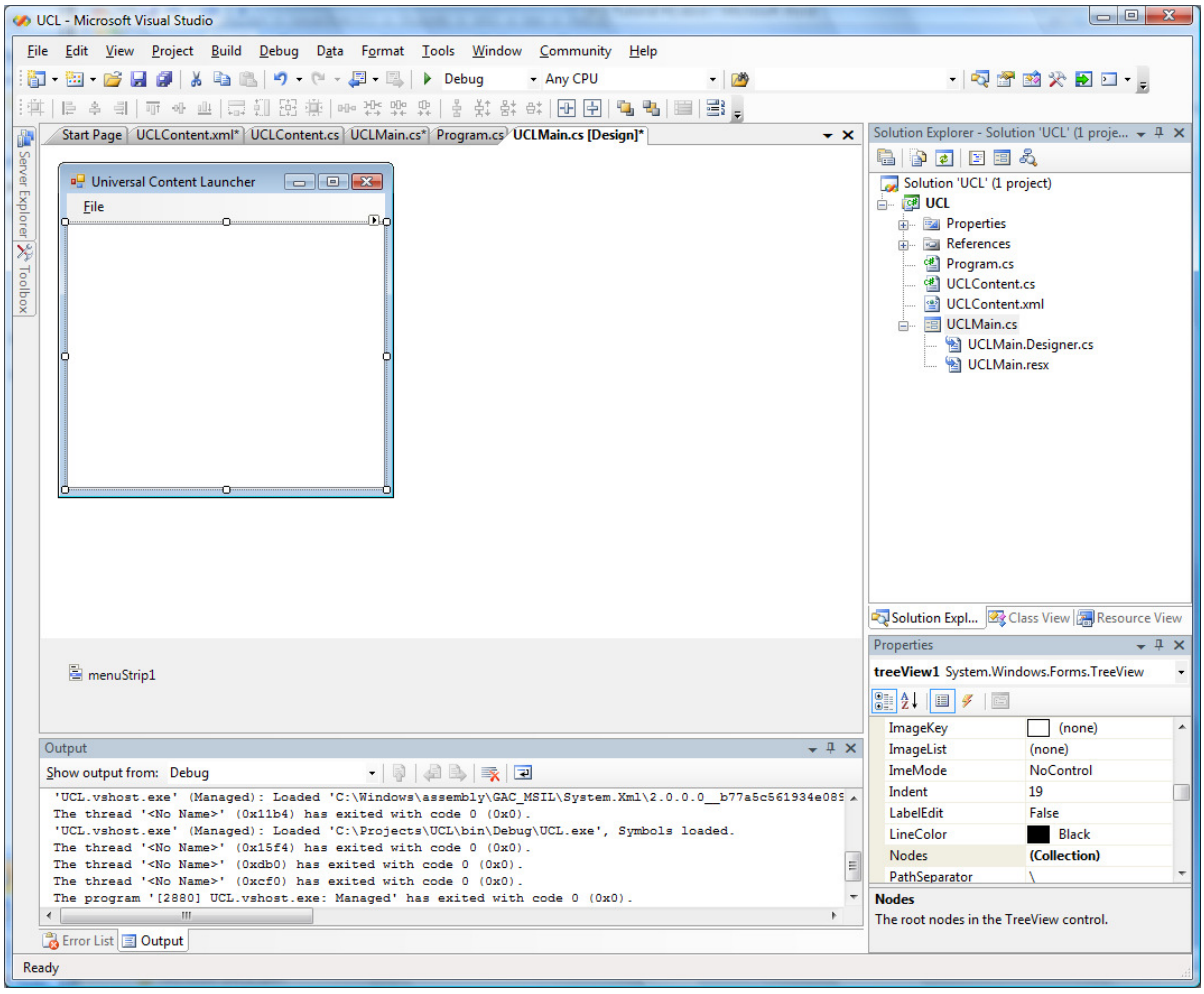
To make this crystal clear, you need to end up with the same lines above, but the "Name", "ExeName" and "ExePath" will match an application, and its location, on your computer.

Once you have created UCLContent.xml, please remember to save it before continuing.

Part Three: Using the Data File

Now we need to do a couple of things with UCL to use the new data file structure we have created. First, we need to add a treeview control to our UCLMain form that will eventually display our content list, and second, we need to write some code that can populate the treeview control from the UCLContent.xml file.

So, let's put a new treeview control onto our UCLMain form. Make sure the UCLMain form design is in the main window, then open the toolbox and click and drag the treeview control onto the forms design. Once there, drag it to the top left corner and then use the resize points around it to fit the size of the form. You should end up with a form that looks like this:



That was easy, wasn't it?

Now we need to add the code to our form to populate the treeview control with the information from the XML file. As we need the form populated when it is first displayed, we could use the forms Load() event as this runs when the form is first loaded, however let's think about that!

We don't just want our content list to populate correctly when the form loads, we may also need to refresh it when we add some new content to it, so how do we get it to populate when we start up in a way that will also enable us to refresh the content list whenever we choose?

Well, to do that we simply need to create a method that populates the treeview control. We can then call the method from our form Load() event, and we will also be able to call it whenever we want to from within the rest of our code. So, let's create a new method called `LoadContentList()`

Go to the UCLMain code file in the main window. First we will add two more `using` statements, to enable us to write the XML Loading method in a clear and understandable way. At the top of the code, add these two new `using` statements below the other `using` statements that are already there.

```
using System.Xml;
using System.Xml.XPath;
```

Now scroll down the code and start to type in the new method directly below the `exitToolStripMenuItem_Click` method. Type in the following lines of code:

```
private void LoadContentList()
{
}
```

Wow, our first method! That's quite simple isn't it? And I know what you're thinking – what does `private void` mean? You've probably noticed them in the other methods that the IDE created for us, but why did we have to also type them in here?

Well, the `private` statement tells C# that this is a private method of our form UCLMain. A private method is a method that can only be seen and used by the form itself, so if you were to list the forms methods and properties, `LoadContentList()` would not be exposed in the list. This is fine in this example, as this method will only ever be used from within its own form, but if you're really on the ball, then it may start you thinking about potential problems with that class we wrote in part one? We'll talk more about this later.

Now, C# expects each method that you write to return a value, while this is useful for some methods, e.g. a method to calculate and return the size of something, or a method to convert the current time to a different time zone, we don't always want to do that. The `void` statement tells C# that we do not expect this method to return a value. If you've done programming before in another language, then you may relate a method that isn't of type `void` to being similar to a function, and a method that is, as being similar to a subroutine.

So, let's fill in the actual code for reading the XML file and populating the treeview, I'm afraid this is where it starts to get a little heavy, but it will lighten up again before the end of this tutorial. Because this is more complex than anything else we have looked at, first I'll show you what we will be typing in, and then we'll go through it line by line.

Before we do that though, it's worth mentioning that the XML file structure was designed to store data. This means that a lot of the methods we can use to retrieve the information in an XML file are similar to those you would use to query data in a database. If you have never used a database before, then this won't help you, but if you have, then you can take comfort from that!

```
    private void LoadContentList()
    {

        XPathDocument doc = new XPathDocument("UCLContent.xml");
        XPathNavigator nav = doc.CreateNavigator();

        treeView1.Nodes.Clear();

        TreeNode tNode;
        string tType = "Application";
        tNode = treeView1.Nodes.Add(tType);

        XPathExpression expr;
        expr = nav.Compile("/Content/" + tType + "/Name");
        XPathNodeIterator iterator = nav.Select(expr);

        try
        {
          while (iterator.MoveNext())
          {
             XPathNavigator nav2 = iterator.Current.Clone();
             tNode.Nodes.Add(nav2.Value);

          }
        }
        catch(Exception ex)
        {
           Console.WriteLine(ex.Message);
        }
}
```

Short but probably confusing! And just when you thought it was going to be easy! Well, let's start at the beginning and go through this code. Note that this code will probably be modified a lot as we go through the tutorials. Therefore, I've made some decisions now that will be of use at a later date, I will explain more about these as we go.

```
XPathDocument doc = new XPathDocument("UCLContent.xml");
XPathNavigator nav = doc.CreateNavigator();
```

The first line creates a new object called `doc` that is of the type `XPathDocument` from out of the `System.Xml.XPath` namespace, and then loads our test XML document into it.

The second line creates a new object called `nav` that is of the type `XPathNavigator`, and then sets itself as a navigator for the XML file we have loaded into the `doc` object. A navigator is a special XML object that allows us to step through, or navigate, an XML file.

```
treeView1.Nodes.Clear();
```

Unless you changed the name of the treeview control when you added it to the UCLMain form, then its name will be `treeView1`. A treeview display allows you to expand and collapse objects called nodes – the most common example of this is the windows explorer that allows you to view the files and folders on your computer via a treeview interface. It's pretty easy

therefore to see that `treeView1.Nodes.Clear();` clears out the entire node list of the control, leaving it empty and ready for us to populate. Remember, this is necessary because we want to be able to call this method to refresh our content list, and if we didn't clear the list first then we would end up appending the same list, over and over, every time we refreshed it.

```
TreeNode tNode;
string tType = "Application";
tNode = treeView1.Nodes.Add(tType);
```

Here are some more objects that we've declared to use in this method. The first one is simply an object called `tNode` that is of the special type `TreeNode`. We need this object, as we need to enter data into a treeview control in a specific way to allow the "tree" structure (the nodes) to display correctly. Therefore, we have created `tNode` so that we can then create treeview child nodes under their parent node, rather than all the nodes being created at the same level, which would end up just being a list! If you are lost, don't panic! This will become more apparent further on.

The second object in the above code is a `string` that contains the text `"Application"`. We need to reference "Application", as, if you remember, that is the element name of the data that we entered into our test XML file. Look back at the UCLContent.xml file if you can't remember.

But if we look at the next line, why didn't we just type `"Application"` Into that line instead, so that it read `tNode = treeView1.Nodes.Add("Application");`? It's only a text string after all, and we wouldn't have needed that extra `string tType` then, would we? Well, that's right, but the reason we've done it this way is because we are starting to think ahead!

Sure we need to create an Application node in the treeview that lists all our applications, the above code will handle this well, but we will also possibly need to create a Game node, Video node and Audio node as well. Do we really want to write the same code out four times if all that will change is the text `"Application"` to `"Game"` for example? And what about in the future, when we expand our `UCLType enum` to include more types? We'll need to do this over and over again – it doesn't bear thinking about really!

By storing the nodes name (which is also incidentally the `UCLType` of the content – more about this in a later tutorial) we can reuse all the code we write by simply putting it in a loop and changing the variable `tType` to each of the subsequent types. So, we can write one piece of code to populate the list for all of our different types – that will be a handy thing to do later on!

```
XPathExpression expr;
expr = nav.Compile("/Content/" + tType + "/Name");
XPathNodeIterator iterator = nav.Select(expr);
```

Here we create another new object called `expr` that is of the type `XPathExpression`, expressions can be used along with the navigators to tell them where they need to be looking to retrieve the required data. From the above code you can see that we create an object for the expression, and then initialise the expression using the navigator we created

earlier, so it is ready to search through the XML file and retrieve a particular element. Note we have again used the `tType` variable to make our code re-useable for any other `UCLType` as well, but this really equates in this instance to:

```
expr = nav.Compile("/Content/Application/Name");
```

Now if you look at the structure of the test XML file we created, you will see that we started off with Content which then contains Application and each Application has a Name. Is this starting to make sense?

Once an expression is linked to a navigator in this way, it can be used to return a result set of all the elements that match the `expr`, in this case we will return all `Application` elements with a `Name`. In order to step through the results, we have defined the object `iterator` which is of type `XPathNodeIterator`. After the `nav.Select(expr)` statement has been executed by C#, then our `iterator` will hold all of the content in our XML file that met the criteria we defined in the `expr` object. I hope your still with me here!

Once we have all that data, then all we have to do now, is loop through it until we reach the end, during which time we can be adding the content description to our treeview node list. This will in turn populate the treeview control, and actually display the content names that are held in our test XML file, within our form. The following section of code does just that, but there are some special things of interest here:

```
try
{
  while (iterator.MoveNext())
  {
     XPathNavigator nav2 = iterator.Current.Clone();
     tNode.Nodes.Add(nav2.Value);

  }
}
catch(Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Firstly, in C# we can use the `try` statement to try and execute a block of code. If this code fails at any point then we can trap the error with a `catch` that must immediately follow the `try` statement. In the example above, we catch any `Exception` (error) in a new object called `ex`. We can then display the `Message` contained within `ex` in the IDE console window, which is a window that exists when we are debugging our application, to let us know exactly what went wrong.

This is great if we want this information back about our program only when running in a debug environment, but actually, the user of our program may want to see this information as well, and they won't be using a debugger! We don't want to show them the actual error `ex.Message` (it will probably blow their mind!) so let's popup a message box with our own custom message.

After the line:

```
Console.WriteLine(ex.Message);
```

Add the line:

```
MessageBox.Show("Cannot load the UCL content list. There is a problem processing the UCLContent.xml file!");
```

This way, we get a detailed error in our debug screen, and they get a simplified error that is easier for them to understand, popping up in a standard windows message box. If there are no errors, then the code in the `try` section will run and the `catch` statement will not be used at all, so we won't see any debug errors, and the user won't see the message box.

The `try` statement above contains a `while` loop to move through the results contained in our `iterator` object. The `while` loop will continue to loop until its condition is no longer true, so in this example the loop is valid until there isn't a `Next` record in the `iterator` to move to.

```
XPathNavigator nav2 = iterator.Current.Clone();
tNode.Nodes.Add(nav2.Value);
```

These are the final lines to discuss in this method – see that wasn't so bad, was it? Here, we are creating a second `XPathNavigator` object called `nav2`, which contains a copy (`Clone`) of the current `iterator` record in the loop. This will then give us a really easy way to access the actual data that the `iterator` data set holds and display it in the treeview.

The final line actually adds the value that is held in the element "Name", to the treeview control. The `iterator` contains the element called "Name" because that is what we told it to search for when we defined the query with `expr`. By writing `nav2.Value`, we are writing the value stored in the element "Name". If we wrote out `nav2.Name` instead, then we would be writing out the text "Name" as that is the Name of the element that we searched for.

An important thing to note here is that we are adding a node to the `tNode` object and not the treeview itself. This allows us to add nodes at the next level of the `tNode` node, so we will have an "Application" Node, and "Microsoft Notepad" will belong to this node. "Application" is referred to as the Parent of "Microsoft Notepad", and "Microsoft Notepad" is a Child of "Application".

If we had just used the following line:

```
treeView1.Nodes.Add(nav2.Value);
```

Then we would have ended up with "Microsoft Notepad" being at the same level in the treeview as "Application", we would not have created the Parent / Child relationship that makes a treeview so powerful when used to display information.

If this confuses you, then try playing about with the code, by changing:

```
tNode.Nodes.Add(nav2.Value);
```

To:

```
treeView1.Nodes.Add(nav2.Value);
```

To see exactly what I mean by this. You should only do this after you verified that the code works correctly of course, so continue to read on, and play around with it later!

That's a lot of talking up above, and hopefully you got a rough feel for how the code works. XML is not the simplest of data structures to use, but it is worthwhile learning about it, and it has a multitude of different uses, as we will see later.

Now we have created our method for populating the treeview control, we need to add it to the UCLMain forms Load() event so that it is run when our program starts, therefore displaying our content list immediately. To do this, make sure the IDE's main view is showing your UCLMain form design and click on the forms title bar to select it.

Now look at the forms property window. Directly above this are a set of icons, the third one along shows the properties for the form (hover your mouse over it, and it will tell you this via a tooltip), and the fourth button along (a lightning bolt) shows the events for the form. Click on the events button and the properties list will change to show you all the available events that your UCLMain form can have.

Scroll down until you see "Load" and double click in the blank field to the right of it. The IDE will now create a new method for you and display it in the code window, just like it did when we were adding the exit menu item in tutorial one. You will see the text pointer flashing between the braces for the following method, eagerly awaiting your input:

```
private void UCLMain_Load(object sender, EventArgs e)
{
}
```

Type the following between the braces:

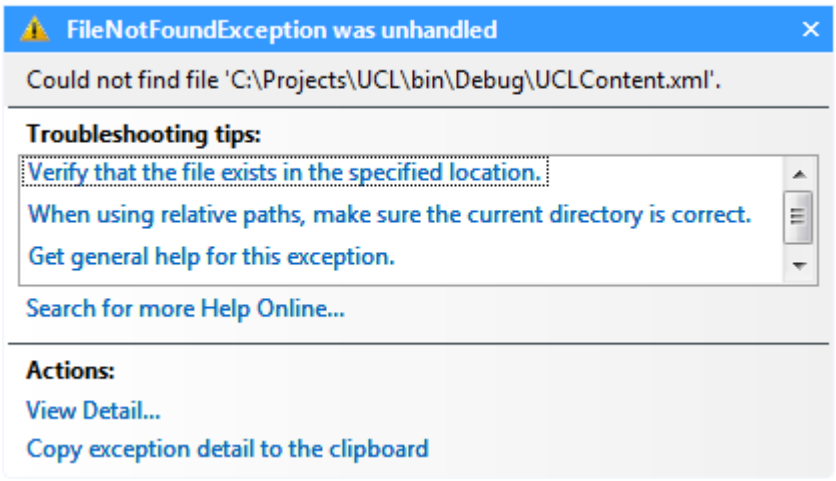
```
    LoadContentList();
```

So you end up with the complete method like so:

```
private void UCLMain_Load(object sender, EventArgs e)
{
    LoadContentList();
}
```

That's it! Our new method `LoadContentList()` will now be called whenever we start up UCL. Ok, it's time to test it, so run UCL and let's see what happens.

Well, you should have got this error:



Ok, stop the project running by clicking on the blue stop square. Notice how much error information the IDE gives back to you here? And most of it can even be clicked on and drilled down further! But, what happened here?

Well, whenever we want to test our project, then the IDE creates an up to date executable of it, so that we can run and debug it. The debug executable that is created when you click on the green “play” triangle to run your project, is located in the folder: bin\Debug within your project folder.

Now, when we wrote the code to process our XML file, we told it to load the file like this:

```
XPathDocument doc = new XPathDocument("UCLContent.xml");
```

You will notice we did not supply a pathname for the location of the XML file, to do so would be foolish, as whoever uses our program could potentially run it from wherever they want. By not specifying a path to the file, UCL will always look for it in the directory that it is running from, so when we are testing and debugging our program, UCL is looking for UCLContent.xml in the path C:\Projects\UCL\bin\Debug\!!!

We can easily sort this out by making sure we put the file there whenever we run UCL from the IDE. But what if this happens to an end user who accidentally deletes the file while mucking around in the program folder? They won’t be running a debug version of UCL, so they will just get a big nasty windows error on their screen. Not good!

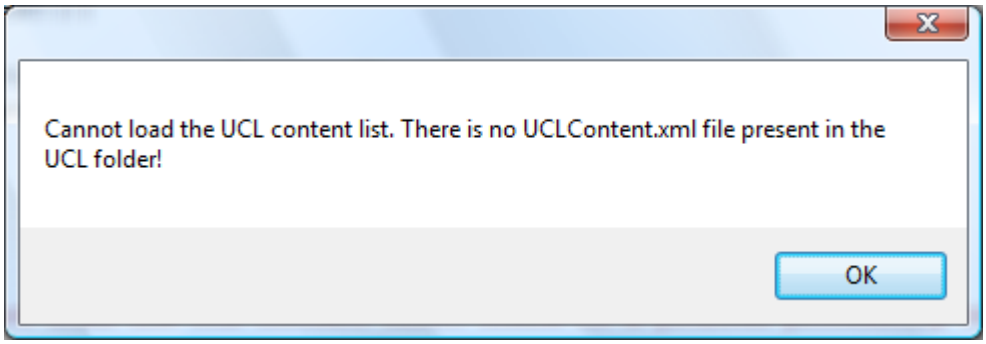
Well, we talked earlier in tutorial one about how we can check if a file exists in C#, can you remember what it was? Below is the code for our `LoadContentList` method after we put in the file check. As well as this code, we also need to add an extra line to the `using` statements at the top of the code, can you guess what it is? I shouldn’t really need to explain all this code now, should I? Have a look and see if you still understand it.

```
        private void LoadContentList()

        {

            if (File.Exists("UCLContent.xml"))
            {
                XPathDocument doc = new XPathDocument("UCLContent.xml");
                XPathNavigator nav = doc.CreateNavigator();

                treeView1.Nodes.Clear();
                TreeNode tNode;
                string tType = "Application";
                tNode = treeView1.Nodes.Add(tType);

                XPathExpression expr;
                expr = nav.Compile("/Content/" + tType + "/Name");
                XPathNodeIterator iterator = nav.Select(expr);

                try
                {
                    while (iterator.MoveNext())
                    {
                        XPathNavigator nav2 = iterator.Current.Clone();
                        tNode.Nodes.Add(nav2.Value);

                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                    MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                }
            }
            else
            {
                MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
            }
        }
```

If you didn't guess what else you needed to add to the `using` statements, then you need to slap yourself hard for not paying attention! You should have added this line to the top of your code:

```
using System.IO;
```

If you run UCL now, you should see the dialogue displayed below, which is much friendlier than a program crash! Also our program will continue to run, even though its content list will be empty, so we could give the user the chance to restore from a backup list etc. Trapping errors like this will help to make our program extremely user friendly.



Great, now that's sorted we need to get back to the original problem, which was the XML file not present in the debug folder. It is actually ok that the file is assumed to be in the same folder as the UCL executable for our final Launcher, this is the preferred way for it to work, but when we're developing our project then we need to make sure that the XML file is located in the same folder as the debug executable or this error will always raise its ugly head and bite us!

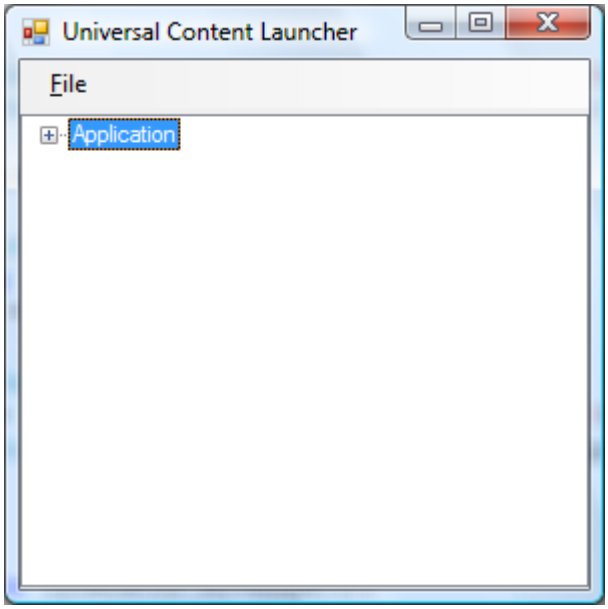
To fix the problem, we just need to make sure there is a copy of the UCLContent.xml file in the bin\Debug folder, but before you start scrambling for windows explorer to copy and paste the file – there is an easier way! If we just copied the file in, then we would need to do this whenever we change the file in our IDE, surely the IDE can handle something as simple as this?

Of course it can!

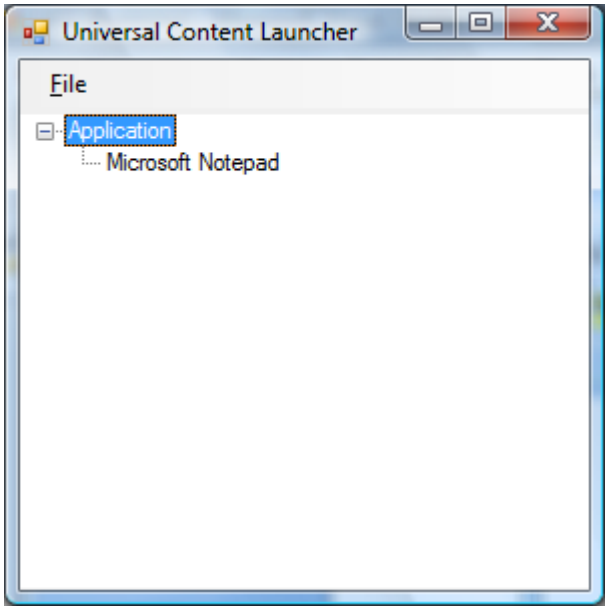
Go to the solution explorer and click on UCLContent.xml and then look in the property window. Scroll down until you see the property "Copy to Output Directory", you can select one of three choices for this property:

- Do not copy
- Copy always
- Copy if newer

Select “Copy if newer” and then run UCL again. You should see the following:



And if you click on the “+” symbol next to Application, you should see this:



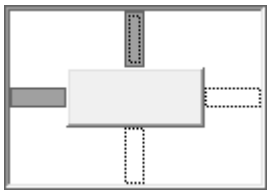
Phew! We got there in the end! But, before we finish this tutorial and get blind stinking drunk to celebrate, let’s do a couple of easy, but worthwhile things:

While your application is running, drag the bottom right corner of the UCL window to make it bigger. You will notice that the UCL form will resize, but the treeview inside it remains the same size.

That looks nasty – let’s fix it up!

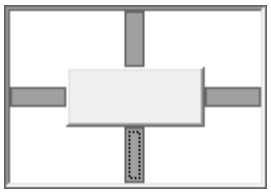
Stop UCL running and go back into the UCLMain form design. Click on the treeview control and look at the properties. Make sure you are viewing the properties, as you may still be viewing the events, in which case you need to click on the property button above the property window, to see the properties list again.

Scroll down until you see the Anchor Property and click on it. Your property display should look like the one below:



This shows you where the current control (treeview1 in this case) is anchored to its parent container - the UCLMain form. At the moment, it is the top left. If we click the right and bottom lines, then our treeview control should be anchored to all sides of the form and grow and shrink with it accordingly.

Change the Anchor property to this:



And then run UCL again.

Marvellous! One last thing though. I don't particularly like the sunken effect around the border of the treeview control, especially now it resizes correctly with the UCL window, it's so last year! So let's get rid of the border.

Once more in the treeview properties, scroll down until you see BorderStyle and change it to "None". Now run UCL for one last time, and sigh in relief at a job well done!

I think we deserve a break. If you got through this and more or less understood what I was waffling on about, then you're well on your way to coding in C#. Take the time between this and the next tutorial to go through what we have done here and try and understand what has happened.

We have learned how to create a class file, and how to create a custom `enum` type. We have created a method, and learned how to call it correctly, and last but by no means least, we have learned how to create an XML data file, read its contents into our program, and populate a treeview control in our main display!

In the next tutorial, we'll actually get to launch some content, and while we're doing that, we'll discover exactly why our UCLContent class doesn't quite cut the mustard!